UCRL-CONF-212607

Tiling Models for Spatial Decomposition in AMTRAN

J. C. Compton, C. J. Clouse

May 31, 2005

Join Russian-American Five-Laboratory Conference on
Computational Mathematics/Physics
Vienna, Austria
June 19, 2005 through June 23, 2005

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Tiling Models for Spatial Decomposition in AMTRAN

John Compton and Christopher Clouse

Lawrence Livermore National Laboratory
7000 East Avenue, Livermore, California, U.S.A.

Effective spatial domain decomposition for discrete ordinate (S_n) neutron transport calculations has been critical for exploiting massively parallel architectures typified by the ASCI White computer at Lawrence Livermore National Laboratory. A combination of geometrical and computational constraints has posed a unique challenge as problems have been scaled up to several thousand processors. Carefully scripted decomposition and corresponding execution algorithms have been developed to handle a range of geometrical and hardware configurations.

Introduction

The AMTRAN code has been in development at Lawrence Livermore National Laboratory since 1995 to solve two- and three-dimensional deterministic neutron transport problems on a range of platforms from serial to massively parallel. When the White computer was delivered to Livermore several years ago as part of the U.S. Department of Energy's Advanced Simulation and Computing Program, there was a concerted effort to adapt various physics codes to exploit its thousands of parallel processors and its threaded, message-passing software environment (IBM, 2000). AMTRAN, in particular, has been able to exploit parallelism in several ways, but spatial domain decomposition has been the most challenging of these, and ultimately the key to the successful and efficient scaling of problems up to thousands of processors.

At the time it began development in 1995, AMTRAN was unique in its application of adaptive mesh refinement technology (AMR) to the solution of the neutron transport equation, although other efforts began appearing in conference proceedings shortly thereafter (Sjoden *et al*., 1996) (Aussourd, 1997). Two basic methodologies have developed concerning the type of AMR used: zone-based (or tree-based) AMR and block-based (or patch-based) AMR (Berger *et al*., 1989). In zone-based AMR, refinement can occur zone by zone, giving greater flexibility in capturing interfaces and gradients with finer zoning, but it does not lend itself to large scale spatial parallelism and can be less cache friendly because of the irregular data layout (Aussourd, 2003). AMTRAN uses a block based AMR, which can result in more total zones than a zone based AMR scheme, but is better suited for large-scale parallelism. AMTRAN blocks are rectangles in 2D, or hexahedral boxes in 3D. Zoning changes are confined to the interfaces between blocks; thus, the zoning within a block is uniform and can be very computationally efficient. Examples of some zoning rules for AMTRAN are illustrated in Figs. 1 and 2. Despite the block AMR structure of AMTRAN, though, the downwind dependency of the Sn directional sweeps still presents a significant challenge in obtaining good spatial parallel efficiency.

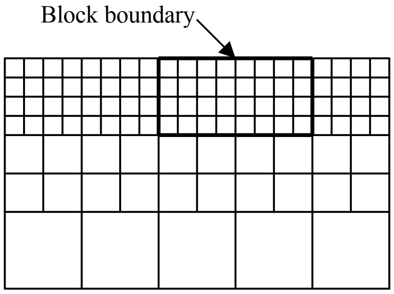


Figure 1. Block boundaries must line up on coarsest grid lines

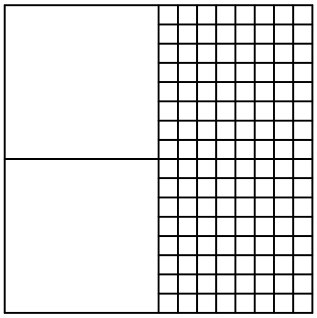


Figure *2.* Zoning changes can be 1→2^n , where *n* is arbitrary

Here we will touch only superficially on those aspects of deterministic neutron transport which have an impact on considerations of computational parallelism and spatial domain decomposition. AMTRAN solves the time-independent transport equation (Eq.1 below), where Ψ_g^m is the angular flux for energy group *g*, and angle *m*.

$$\vec{\Omega} \bullet \vec{\nabla}\Psi_g^m + \sigma_{tot}\Psi_g^m = S(\psi) \qquad (1)$$

The total cross section is represented by σ_{tot} , and $S(\psi)$ is the source term representing contributions from other angles and energy groups through scattering as well as fission sources. $S(\psi)$ is typically only a function of the scalar flux, defined as a weighted sum of the angular fluxes, and its moments. The transport problem is computed on a Cartesian finite-element grid. It consists of one or more uniform blocks. The zoning in each block is determined by the properties of the physical system being modeled (in particular, to an approximation of the local mean free path of the neutrons through the medium). The neutron transport equations are then solved iteratively until the degree of convergence specified by the user is reached. That is the broadest explanation possible to describe the essence of the method. Two immediate sources of parallelism lie in the fact that (1) the time-independent transport equation (Eq. 1) is solved using a set of coupled single-energy group transport equations, each of which can be solved in parallel, subject to synchronization points where the coupling terms are calculated, and (2) that the transport equation is further discretized into angles, each of which may likewise be computed independently. In our implementation the energy groups (typically 6 to 48) are distributed among the processors, and the angles are distributed among the threads (each of which is here supported by a separate processor sharing a common memory space). These two strategies permit a degree of parallelism equal to the number of energy groups multiplied by the number of computational threads. For practical reasons, threading is limited to the number of angles per octant, or quadrant in 2D (typically 3 to 16), and also by the hardware (typically 2 to 16 processors on a shared memory node). Any further parallelism to be achieved beyond this point requires spatial domain decomposition, and that leads us into the heart of our discussion.

As is typically done in cases of domain decomposition, computation proceeds in parallel for each region while being punctuated by occasional exchanges of information among the processors. In AMTRAN's case, each iteration consists of sub-iterations separated by a computational barrier and exchange of messages among the processors. Exchange is necessary only between pairs of processors which share a common border corresponding to the physical space being modeled. In particular, the messages consist of information describing the physical variables being computed in each region. Since the calculation is relatively static from one iteration to the next, the size and content of the messages is predetermined by the spatial domain decomposition performed initially. However, as we shall see, we may permit the subdomain boundaries to shift dynamically between iterations if load imbalances are detected during the computation.

Basic Definitions

At this point we need to introduce some definitions that will be used throughout the remainder of the discussion. We have already used the term "subdomain" in the preceding paragraphs. To be more precise, a subdomain consists of one or more contiguous blocks or grids which fill a space corresponding to a rectangle (in two dimensions) or hexahedral box (in three dimensions), that is, a subset of the original problem space. In Fig. 3 we have a very simplistic example of four subdomains, where solid lines denote subdomain boundaries, thick dashed lines denote grid boundaries within subdomains, and thin dashed lines separate individual zones within a grid. In this example subdomain 1 contains the single grid A, subdomain 2 contains grid B, subdomain 3 contains the three grids C, D and E, and subdomain 4 contains grids F and G. (Typically grids contain hundreds or thousands of zones.)

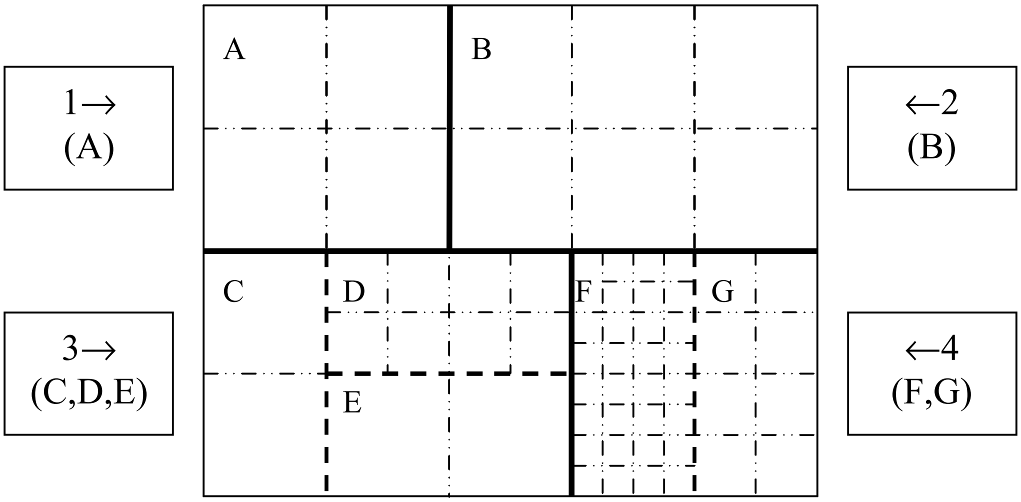


Figure 3. Example of four subdomains with seven grids

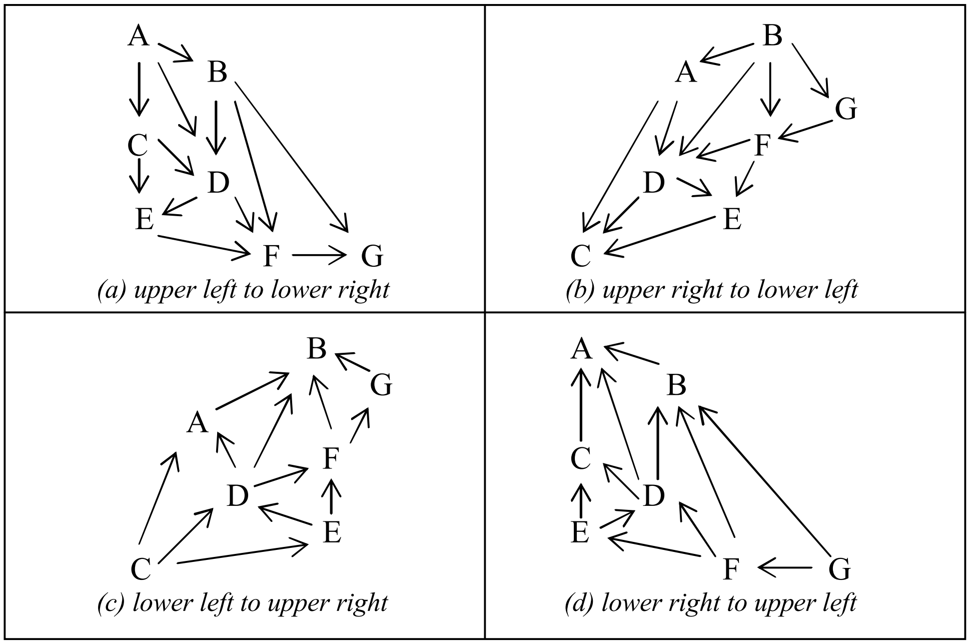
In our terminology a "subdomain master" is a collection of one or more subdomains that are assigned to a given processor. In addition, if a master contains more than one subdomain, then we speak of this as an example of "subdomain overloading." These subdomains may or may not be contiguous, so that a subdomain master may represent space physically distributed throughout the problem (namely, as a collection of rectangles or hexahedral boxes). Thus subdomains are individually physically coherent (in terms of the object being modeled), whereas master subdomains are logically coherent. The only physical coherence that the latter need possess is the fact that they reside in adjacent computer memory. Returning to the example of Fig. 3, we could place subdomains 1 and 4 under a single master, and leave subdomains 2 and 3 each under a separate master.

Note again that a subdomain corresponds purely to a single physical space. This means that even though each subdomain is replicated across multiple processors (each corresponding to one or more separate energy groups), it is never split between processors. In summary, sets of processors are partitioned among subdomain masters and, within a subdomain master set, processors are partitioned by energy group. Each member of the set models the same physical space and therefore behaves quite similarly in terms of the computations it performs and the structure of the messages it sends and receives. The term "master" arises from the fact that one designated processor performs certain collection and messaging functions both within the set (such as with all-reduce functions) and then later with the other sets. We can use the example in Fig. 3 to illustrate how subdomains, grids, and energy groups could be allocated among processors. Let us say that there are four energy groups (a)-(d), with two each assigned to a processor. Table 1 gives the assignments assuming, as we did above, that subdomains 1 and 4 belong to a single master.

Table 1. Example of subdomain-to-processor assignment

Processor	Master	Subdomains	Grids	Energy groups
1	1	1,4	A,F,G	a,b
2		1,4	A,F,G	c,d
3	2	2	B	a,b
4		2	B	c,d
5	3	3	C,D,E	a,b
6		3	C,D,E	c,d

A final bit of terminology is crucial to our discussion and needs special attention. This concerns the discretized angles mentioned above and the order of computations. Eq. 1 is solved through "source iteration" in which the RHS source term is evaluated using "old" values of the fluxes; usually those from the previous iteration. The streaming term on the LHS is then solved by sweeping through the mesh in the direction of neutron flow, given by the direction of angle *m*. In two dimensions the sweep angles can be assigned to one of four groups based upon the direction of the sweep from one of the four corners of the problem to the corresponding opposite corner (that is, lower left to upper right, lower right to upper left, upper left to lower right, or upper right to lower left). Each one of these sweeps is independent of the others and so they can be done in parallel. However, within a single sweep, all the grids must be swept with certain dependency rules that govern the sequence of processing of grids in going from the grid at one corner and ending with the grid in the opposite corner. We will use the sweep direction from lower left to upper right to illustrate the order of computation between these two extremes. The basic rule is that each boundary on the bottom or left of a subdomain must either be an exterior boundary, or else be adjacent only to subdomains that have already been swept. All four sets of sweep dependencies corresponding to the example of Fig. 3 are given in Fig. 4. It should be noted from the diagrams that dependencies for sweeps (a) and (d), which proceed in opposite directions, are in fact inverses of one another, which can be seen by reversing the direction of the arrows. The same holds for cases (b) and (c) and illustrates that dependencies for sweeps in opposite directions are always inverses of one another.



(a) upper left to lower right

(b) upper right to lower left

(c) lower left to upper right

(d) lower right to upper left

Figure 4. Example of sweep dependencies for seven grids in four subdomains

In three dimensions we have the analogous situation with regard to sweeps, except that all angles fall into one of eight octants to begin their sweep, starting from the outermost corner of the octant and ending in the opposite corner of the problem. Now in order for a subdomain to be swept (i.e., the upwind sides) it must have the three sides facing the sweep corner either lying on one of the external boundary planes of the problem or else adjacent only to subdomains previously swept from the same direction. Examples below will illustrate these dependency rules.

Motivation by Example for the Decomposition Method

We now turn to the problem of how sweeps are actually performed on each processor once we have partitioned the problem among subdomains, assigned these to masters, and performed adaptive mesh refinement to generate the grids on which the computation will be performed. This will illustrate how critically the computational process depends on both the initial partitioning of the problem, and on how we choose among the various possible ways of scheduling the potentially parallel subtasks assigned to a given processor. Later we will give solutions that address all these concerns.

Let us use our previous example from Fig. 3, but unlike the case of Table 1, each subdomain will have a separate master, so that there will be eight processors in total instead of six. (This assignment to masters has no effect on the dependency diagrams of Fig. 4.) We will also assume that the total work to be performed in each subdomain is equal, that is, that they are perfectly load balanced in terms the amount of computation which they must perform. A simple strategy is for each processor to perform all sweeps possible for a given sub-iteration, then exchange messages as necessary, and proceed to the next sub-iteration until all sweeps have been performed in all directions on all processors. (Recall that messaging must occur in order to pass information across grid boundaries.)

Referring to the dependency diagrams of Fig. 4, we see that on the first sub-iteration grid A can be swept in subdomain 1 from the upper left, grid B in subdomain 2 from the upper right, grids C, then E, then D in subdomain 3 from the lower left, and finally grids G and then F in subdomain 4 from the lower right. At this point nothing else can be done until the processors have exchanged messages. The second sub-iteration is not quite so straightforward. Grid A in subdomain 1 can now be swept both from the lower left and from the upper right. However, grid B in subdomain 2 can be swept only from the upper left. (It cannot be swept from the lower right because grid D has not yet been swept in that direction.) Likewise, only grid C in subdomain 3 can be swept from the upper left, whereas E, then D, and then C can all be swept from the lower right. In subdomain 4, grid G and then F can be swept from the upper right, and in the opposite order from the lower left. These and the remaining sub-iterations giving the complete set of sweeps for a single iteration are listed in Table 2. We have assigned arbitrary time units to each grid so that the total for each subdomain is one unit.

Table 2. Order of computations for the greedy algorithm

Processors and Subdomains→	Processors 1 and 2 Subdomain 1			Processors 3 and 4 Subdomain 2			Processors 5 and 6 Subdomain 3			Processors 7 and 8 Subdomain 4		
Sub-iteration	Grid	Sweep	Time	Grid	Sweep	Time	Grid	Sweep	Time	Grid	Sweep	Time
1	A	ul→lr	1.00	B	ur→ll	1.00	C	ll→ur	.25	G	lr→ul	.50
							E		.25	F		.50
							D		.50			
2	A	ur→ll	1.00	B	ul→lr	1.00	C	ul→lr	.25	F	ll→ur	.50
							E	lr→ul	.25	G		.50
	A	ll→ur	1.00				D		.50	G	ur→ll	.50
							C		.25	F		.50
3	(idle)			B	lr→ul	1.00	D	ur→ll	.50	(idle)		
							E		.25			
				B	ll→ur	1.00	D	ul→lr	.50			
							E		.25			
4	A	lr→ul	1.00	(idle)			C	ur→ll	.25	F	ul→lr	.50
										G		.50

Note: ul = upper left, ur = upper right, ll = lower left, lr = lower right

It is clear from a glance at Table 2 that the strategy of doing all computation possible on every sub-iteration is far from optimal. First we see that different processors do comparable work only in the first sub-iteration. After that we find great asymmetry and idled processors. In order to quantify this imbalance we can take the maximum time for any processor on a sub-iteration as the time required to complete that sub-iteration across all processors (at which point they exchange messages). What we see from the table is that the four sub-iterations take (1, 2, 2, 1) time units, respectively. This gives an overall time of 6 units (where we ignore communication time among the processors). Let us now take another approach and show how we can obtain far different efficiencies by a different scheduling of computations for the same problem. Table 3 gives the result.

Table 3. Order of computations for optimized strategy

Processors and Subdomains→	Processors 1 and 2 Subdomain 1			Processors 3 and 4 Subdomain 2			Processors 5 and 6 Subdomain 3			Processors 7 and 8 Subdomain 4		
Sub-iteration	Grid	Sweep	Time	Grid	Sweep	Time	Grid	Sweep	Time	Grid	Sweep	Time
1	A	ul→lr	1.00	B	ur→ll	1.00	C	ll→ur	.25	G	lr→ul	.50
							E		.25	F		.50
							D		.50			
2	A	ur→ll	1.00	B	ul→lr	1.00	E	lr→ul	.25	F	ll→ur	.50
							D		.50	G		.50
							C		.25			
3	A	ll→ur	1.00	B	lr→ul	1.00	C	ul→lr	.25	G	ur→ll	.50
							D		.50	F		.50
							E		.25			
4	A	lr→ul	1.00	B	ll→ur	1.00	D	ur→ll	.50	F	ul→lr	.50
							E		.25	G		.50
							C		.25			

Note: ul = upper left, ur = upper right, ll = lower left, lr = lower right

Now we find a completely different situation in comparison with Table 2. We see in particular that each processor does one unit of work on each sub-iteration. Thus the overall time has been reduced from 6 units to 4 for the complete iteration, a savings of 50%. Furthermore we see that all the grids of a subdomain are swept in each sub-iteration, a situation that does not always hold in Table 2. This was possible because (1) we had precisely four subdomains, (2) the space was partitioned equally among the subdomains in advance, and (3) there existed dependency relationships among the subdomains that permitted this unique distribution of effort among the sub-iterations. Note that by sweeping all the grids in a subdomain in only one direction per sub-iteration, we need not be concerned about the internal structure of the subdomain, but can treat each subdomain as essentially one super-grid. Then our precedence rules (such as those illustrated in Fig. 4) become greatly simplified. (There is a practical consideration here as well, in that we must perform the domain decomposition before the stage of adaptive mesh refinement is begun. This means that we do not know where the grid boundaries will lie at the time of the decomposition. Thus we wish to avoid coupling the internal grid structure of the subdomains to our decomposition strategy.)

The approach described above forms one of the cornerstones of AMTRAN's domain decomposition strategy, which covers not only the simplest possible case examined in the preceding paragraphs, but up to 512 subdomains in three dimensions. In order to accomplish this task AMTRAN internally maintains tables which drive the domain decomposition procedure on problem initialization, and additional corresponding tables of scripts for deciding which angles to sweep on each sub-iteration as well. Each handles a fixed number of subdomains (as requested by the code user) in either two or three dimensions. This strict regime reduces the number of possible cases that we need to analyze and guarantees uniformity of behavior within the code.

Outline of the Decomposition Algorithm

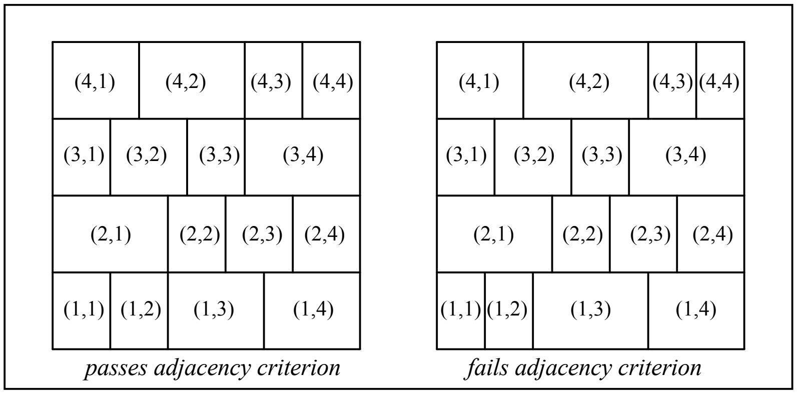
We have now provided the motivation for our spatial decomposition strategy, which we now give in greater detail and generality. We have found it useful so far to support 4, 8, or 16 subdomains in two dimensions, and 8, 16, 32, 64, 128, 256, or 512 subdomains in three. The reason that we support only powers of two in this way is that they permit the most efficient exploitation of the parallel computational environment available to us. Our partition algorithm is simple and rigid. We make a fixed number of binary partitions along one axis, then for each subdivided region we make binary partitions along a second axis, and then in the case of three dimensions along the remaining axis. A weighting scheme that estimates the amount of work to be performed in each part of the problem is the driver behind the initial set of partitions. (As stated earlier, we can later rebalance the problem dynamically stepwise after each partition in case the initial approximation is not adequate in terms of load balance.) The choice of which Cartesian axis is used in each set of partitions depends upon the presence or absence of symmetry planes along each axis and is beyond the scope of this discussion. Table 4 gives the binary decomposition specifications that AMTRAN uses for each configuration of subdomains.

Table 4. Order and degree of partitioning according to number of subdomains

Number of Dimensions	Number of Subdomains	Number of Regions Partitioned along Axis 1	 Axis 2	 Axis 3
2	4	2	2	
2	8	2	4	
2	16	4	4	
3	8	2	2	2
3	16	2	2	4
3	32	2	4	4
3	64	4	4	4
3	128	8	4	4
3	256	8	8	4
3	512	8	8	8

With the binary partition scheme above we can now name our subdomains by their coordinates along each partition axis, and dispense with the individual numbers used earlier. By way of illustration, the sample problem of Fig. 3 was partitioned first along the y-axis into what will become subdomains 1 and 2 on top and subdomains 3 and 4 on the bottom. Then we separately partition the upper subdomain into 1 and 2 and the lower into 3 and 4. So in the new subdomain coordinate numbering system subdomain 1 becomes subdomain (1,1), subdomain 2 becomes subdomain (1,2), subdomain 3 becomes subdomain (2,1), and finally subdomain 4 becomes subdomain (2,2). We will use this latter numbering in the remainder of our examples.

We now want to formalize our criteria for sweeping subdomains monolithically as in Table 3 above. Note there that in the first two sub-iterations, the upper two subdomains exchanged sweep directions, and the lower two did likewise. Then in the last two sub-iterations, the upper two subdomains swapped the pattern of the first two sub-iterations with the lower two subdomains. This is in fact the only pattern of sweeps which can give the optimal performance and illustrates the fact that sweeps directions are exchanged among subdomains according to the *inverse* order of partitioning. We can alternatively view this as exchange first between nearest neighbors on the binary partition tree, then among progressively more distant branches on that tree. (We will expand on this topic with more examples in the next section.) The second criterion for monolithic sweeping of subdomains, as we have already mentioned in earlier discussion, is to sweep only in one "direction" per subdomain per sub-iteration, where "direction" in our terminology means all angles lying in a given octant (or quadrant in 2D). We can now introduce the third and final criterion, which we will call the "adjacency" criterion. This is that subdomains which share a common boundary can have subdomain coordinates which differ by no more than one in any axis. It guarantees that the prerequisite sweeps have already been performed for each subdomain's neighbors in time for it to perform its own sweeps. In Fig. 5 we contrast decompositions which both pass and fail this criterion. In the one on the left adjacent subdomains have coordinates which differ by no more that one in either axis, whereas on the right we have shifted the partitions slightly. Now subdomain (2,1) shares a boundary with subdomain (1,3), and subdomain (4,2) shares a boundary with subdomain (3,4). In both cases the second coordinates differ by two, violating the adjacency criterion.



passes adjacency criterion

fails adjacency criterion

Figure 5. A comparison of subdomain adjacency in two decompositions

Before we go into more detail on sweeps and their relationship to domain partitioning, it will be useful to introduce some additional notation.

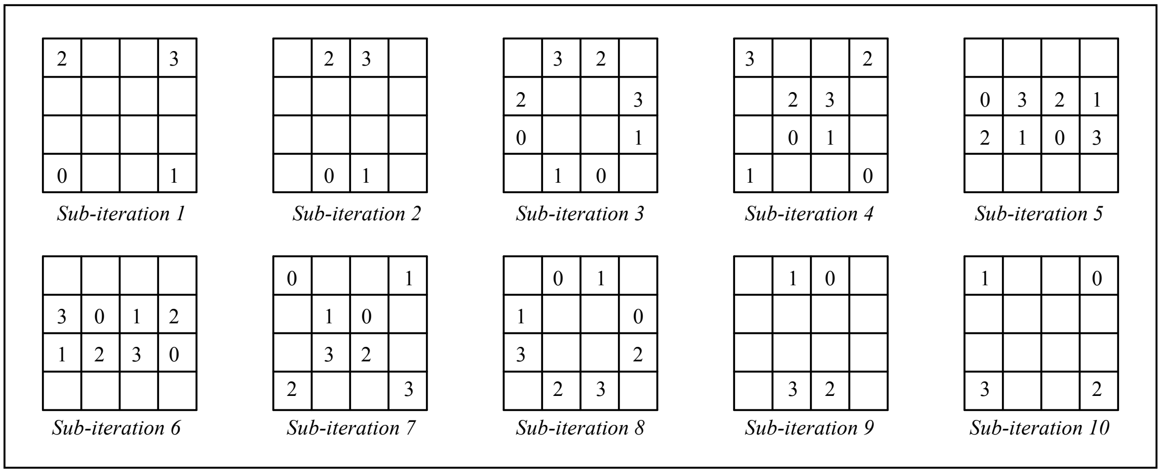
Representing Sweeps and Sweep Directions

We will now introduce a numerical notation for sweep directions and introduce three-dimensional sweeps. Table 5 gives the correspondence. We can use the numerical sweep direction values in a schematic representation of sweeps in a diagram representing those for 16 subdomains in a four-by-four two-dimensional problem. Fig. 6 shows how we might give a more abstract representation of a problem such as that shown in Fig. 5. We do not assume in Fig. 6 that all the subdomain boundaries necessarily align, but merely want a general representation for all four-by-four problems that satisfy our sweep criteria of section 4 above. Ten sub-iterations are represented.

Referring back to Fig. 5, we can see why the adjacency criterion must hold in order for the pattern of Fig. 6 to work. For the set of subdomains on the right side of Fig. 5 we are blocked at sub-iteration 3 from performing the sweep in the 0 direction for subdomain (2,1). This is because subdomain (1,3) needs to have already been swept in direction 0 (and not merely be in the process of being swept as shown in the pattern) for (2,1) to be swept at sub-iteration 3. For the set of subdomains on the left side of Fig. 5, the sweep patterns of Fig. 6 work properly as expected.

Table 5. Numerical representation of sweep directions

Sweep Direction	Number of Dimensions	Interpretation
0	2	lower left to upper right
1	2	lower right to upper left
2	2	upper left to lower right
3	2	upper right to lower left
0	3	lower left front to upper right back
1	3	lower right front to upper left back
2	3	lower left back to upper right front
3	3	lower right back to upper left front
4	3	upper left front to lower right back
5	3	upper right front to lower left back
6	3	upper left back to lower right front
7	3	upper right back to lower left front



2			3
0			1

Sub-iteration 1

	2	3	
	0	1	

Sub-iteration 2

	3	2	
2			3
0			1
	1	0	

Sub-iteration 3

3			2
	2	3	
	0	1	
1			0

Sub-iteration 4

0	3	2	1
2	1	0	3

Sub-iteration 5

3	0	1	2
1	2	3	0

Sub-iteration 6

0			1
	1	0	
	3	2	
2			3

Sub-iteration 7

	0	1	
1			0
3			2
	2	3	

Sub-iteration 8

	1	0	
	3	2	

Sub-iteration 9

1			0
3			2

Sub-iteration 10

Figure 6. Sweep pattern for 16 subdomains in two dimensions

Let us now consider some efficiency issues concerning the sweep patterns of Fig. 6. We notice that each subdomain is swept only four times out of ten sub-iterations for an efficiency rate of only 40%. However, we can double the efficiency to 80% by noticing that subdomains separated by two units vertically are never both swept in the same sub-iteration. So we can perform subdomain overloading and place these pairs of subdomains on single masters. For example, subdomains (1,1) and (3,1) are paired together, (2,1) and (4,1) are paired together, and so on. Thus master 1 will be busy with subdomain (1,1) in sub-iteration 1, subdomain (3,1) in sub-iteration 3, subdomain (1,1) again in sub-iteration 4 again, and so forth. We will find subdomain overloading to be essential to achieving maximal efficiencies as we move toward higher numbers of subdomains as well. Note that 100% theoretical efficiencies are possible only for four subdomains in two dimensions and eight subdomains in three dimensions because otherwise (with more subdomains than sweep directions) at least the initial and final sub-iterations must necessarily idle some processors.

In order to represent sweep patterns more compactly, we can use a chart form as shown in Table 6. It gives the sweep pattern of Fig. 6 both without and with subdomain overloading.

Table 6. Sweep directions for a 4×4 set of subdomains

Master	Subdomain	Sub-iteration										Master	Subdomains	Sub-iteration									
(without subdomain overloading)												(with subdomain overloading)											
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*			*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
		Sweep Directions												Sweep Directions									
1	(1,1)	0			1			2			3	1	(1,1),**(3,1)**	0		**2**	1	**0**	**3**	2	**1**		3
2	(1,2)		0	1					2	3		2	(1,2),**(3,2)**		0	1	**2**	**3**	**0**	**1**	2	3	
3	(1,3)		1	0					3	2		3	(1,3),**(3,3)**		1	0	**3**	**2**	**1**	**0**	3	2	
4	(1,4)	1			0			3			2	4	(1,4),**(3,4)**	1		**3**	0	**1**	**2**	3	**0**		2
5	(2,1)			0		2	1		3			5	(2,1),**(4,1)**	**2**		0	**3**	2	1	**0**	3		**1**
6	(2,2)				0	1	2	3				6	(2,2),**(4,2)**		**2**	**3**	0	1	2	3	**0**	**1**	
7	(2,3)				1	0	3	2				7	(2,3),**(4,3)**		**3**	**2**	1	0	3	2	**1**	**0**	
8	(2,4)			1		3	0		2			8	(2,4),**(4,4)**	**3**		1	**2**	3	0	**1**	2		**0**
9	(3,1)			2		0	3		1														
10	(3,2)				2	3	0	1															
11	(3,3)				3	2	1	0															
12	(3,4)			3		1	2		0														
13	(4,1)	2			3			0			1												
14	(4,2)		2	3					0	1													
15	(4,3)		3	2					1	0													
16	(4,4)	3			2			1			0												

We can use this same technique of overloading for our first three-dimensional example, namely, a 4×4×2 array of subdomains. The problem will consist of two layers, each swept in much the same fashion as above. The schematic view of the first sub-iteration is shown in Fig. 7, where heavy arrows are used to denote sweep directions. In addition, some of the subdomains at the corners of the block have been labeled in order to show their numbering scheme. The complete set of sweeps is given in numerical notation in Table 7. Note that now each subdomain must be swept eight times instead of four times as in the two-dimensional examples which preceded this one. This increases the number of sub-iterations from 10 to 18. We have also shaded the upper left quadrant to highlight where the pattern in the upper right of Table 6 has been replicated four times in Table 7.

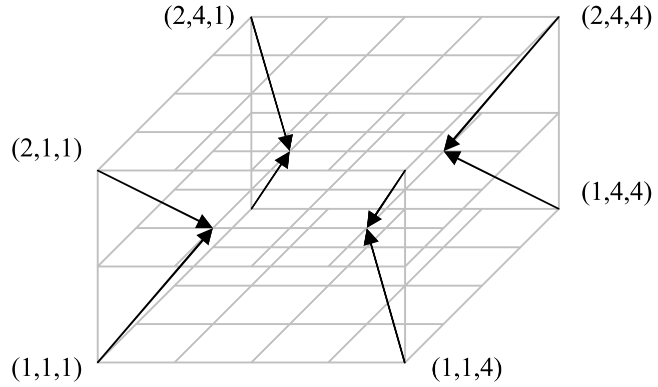


Figure 7. Schematic view of 32 subdomains in a 4×4×2 configuration

Table 7. Sweep directions for a 4×4×2 set of subdomains on 16 masters

Master	Subdomains	Sub-iteration																	
		1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*
		Sweep Directions																	
1	(1,1,1),**(1,3,1)**	0		**2**	1	**0**	**3**	2	**1**	*4*	3	***6***	*5*	***4***	*7*	*6*	***5***		*7*
2	(1,1,2),**(1,3,2)**		0	1	**2**	**3**	**0**	**1**	2	3	*4*	*5*	***6***	*7*	***4***	***5***	*6*	*7*	
3	(1,1,3),**(1,3,3)**		1	0	**3**	**2**	**1**	**0**	3	2	*5*	*4*	*7*	***6***	***5***	***4***	*7*	*6*	
4	(1,1,4),**(1,3,4)**	1		**3**	0	**1**	**2**	3	**0**	*5*	2	*7*	*4*	***5***	***6***	*7*	***4***		*6*
5	(1,2,1),**(1,4,1)**	**2**		0	**3**	2	1	**0**	3	***6***	**1**	*4*	***7***	*6*	*5*	***4***	*7*		***5***
6	(1,2,2),**(1,4,2)**		**2**	**3**	0	1	2	3	**0**	**1**	***6***	*7*	*4*	*5*	*6*	*7*	***4***	***5***	
7	(1,2,3),**(1,4,3)**		**3**	**2**	1	0	3	2	**1**	**0**	*7*	***6***	*5*	*4*	*7*	*6*	***5***	***4***	
8	(1,2,4),**(1,4,4)**	**3**		1	**2**	3	0	**1**	2	*7*	**0**	*5*	***6***	*7*	***4***	***5***	*6*		***4***
9	(2,1,1),**(2,3,1)**	4		**6**	5	**4**	**7**	6	5	0	**7**	**2**	1	0	**3**	2	**1**		3
10	(2,1,2),**(2,3,2)**		4	5	**6**	**7**	**4**	**5**	6	7	0	1	**2**	**3**	**0**	**1**	2	3	
11	(2,1,3),**(2,3,3)**		5	4	**7**	**6**	**5**	**4**	7	6	1	0	**3**	**2**	**1**	**0**	3	2	
12	(2,1,4),**(2,3,4)**	5		**7**	4	**5**	**6**	7	**4**	1	**6**	**3**	0	**1**	**2**	3	**0**		2
13	(2,2,1),**(2,4,1)**	**6**		4	**7**	6	5	**4**	7	**2**	**5**	0	**3**	2	1	**0**	3		**1**
14	(2,2,2),**(2,4,2)**		**6**	**7**	4	5	6	7	**4**	**5**	**2**	**3**	0	1	2	3	**0**	**1**	
15	(2,2,3),**(2,4,3)**		**7**	**6**	5	4	7	6	**5**	**4**	**3**	**2**	1	0	3	2	**1**	**0**	
16	(2,2,4),**(2,4,4)**	**7**		5	**6**	7	4	**5**	6	3	**4**	1	**2**	3	0	**1**	2		**0**

We now wish to expand our scheme for subdomain overloading to larger numbers of subdomains. In order to do this we need to examine more carefully the shaded sweep pattern at the left of Table 6. When we compare it to the shaded pattern in Table 7, we can see that the upper and lower halves of the former pattern have been merged by overlaying one half on top of the other. Furthermore, to the right of the shaded pattern in Table 7 we have placed the same pattern (the unshaded cells with italicized numerals) in an interlocking fashion with the shaded area. This illustrates an interesting property of the pattern that relates to tiling theory, namely, that the shaded pattern of Table 6 can completely tile the plane if displaced in (positive or negative) multiples of 8 units in the vertical and (positive or negative) multiples of 8 in the horizontal direction. That is, copies of the pattern, if treated like the pieces of a jigsaw puzzle, can be placed together to cover a planar surface infinitely in all directions. This replication scheme works not only for modeling the sweeps of our 16 and 32 subdomain configurations, as already shown, but for overloading higher numbers of subdomains onto sixteen masters as well. For example, in order to place 64 subdomains on 16 masters we replicate the sweeps of Table 6 to the right, overlaying the copy of sub-iterations 1 and 2 on sub-iterations 17 and 18, respectively, for a new total of 34=2×18-2 sub-iterations. Master 1 would have subdomains {(1,1,1),(1,3,1),(3,1,1),(3,3,1)} and master 9 would have {(2,1,1),(2,3,1),(4,1,1),(4,3,1)}, for example. We could similarly replicate patterns to go to 128 or 256 subdomains.

Efficiency Comparisons for Various Subdomain Configurations

We can now summarize the various configurations of subdomains and masters that we studied in the previous section. Table 8 contains these results, plus some that were not covered earlier. Again, these are theoretical results assuming perfect load balancing and no overhead or communications cost. It is clear from this table that subdomain overloading is preferable whenever we have more than 4 subdomains in two dimensions or 8 subdomains in three. The asymptotic effect of overloading subdomains on masters is apparent also in going from 1 to 2 to 4 to 8 subdomains per master on 16 subdomains (with efficiencies of 80, 89, 94, and 97 percent respectively). On 32 masters the same overloading produces efficiencies of 57, 73, 84, and 91 percent, respectively. Likewise, for 64 masters we have efficiencies of 36, 61, 76, and 86 percent, respectively. This is shown graphically in Fig. 8.

Table 8. Theoretical efficiency results for various configurations of subdomains

Dimensions	Masters	Subdomains	Subdomains per Master	Active Sub-iterations	Total Sub-iterations	Percent Efficiency
2	4	4	1	4	4	100
2	8	8	1	4	6	67
2	8	16	2	8	10	80
2	16	16	1	4	10	40
3	8	8	1	8	8	100
3	16	16	1	8	10	80
3	16	32	2	16	18	89
3	16	64	4	32	34	94
3	16	128	8	64	66	97
3	16	256	16	128	130	98
3	16	512	32	256	258	99
3	32	32	1	8	14	57
3	32	64	2	16	22	73
3	32	128	4	32	38	84
3	32	256	8	64	70	91
3	32	512	16	128	134	96
3	64	64	1	8	22	36
3	64	128	2	16	26	61
3	64	256	4	32	42	76
3	64	512	8	64	74	86
3	128	512	4	32	50	64
3	128	1024	8	64	82	78
3	128	2048	16	128	146	88
3	128	4096	32	256	274	93

The important observation to be made from Fig. 8 is that for both 16 and 32 masters we can adequately exploit our computational resources by subdomain overloading (with progressively larger numbers of subdomains). Note that it is the number of masters, rather that the number of subdomains, that determines the degree of parallelism overall, so that all data points along a single curve represent the same level. As an example with, say 32 energy groups, 1024 processors can be used with 32 masters. Although 64 masters allow for somewhat less exploitation of our resources, as can be seen from Table8 and Fig. 8, they permit us to use up to 2048 processors (again with 32 energy groups). The optional use of threading increases these figures even more. The last entry in Table 8 corresponds to a 16×16×16 decomposition of the problem space into 4096 subdomains, with 32 subdomains on each of 128 masters. With 32 energy groups and 4-way threading, we would use 16384 processors, a capacity to be reached within a few years.

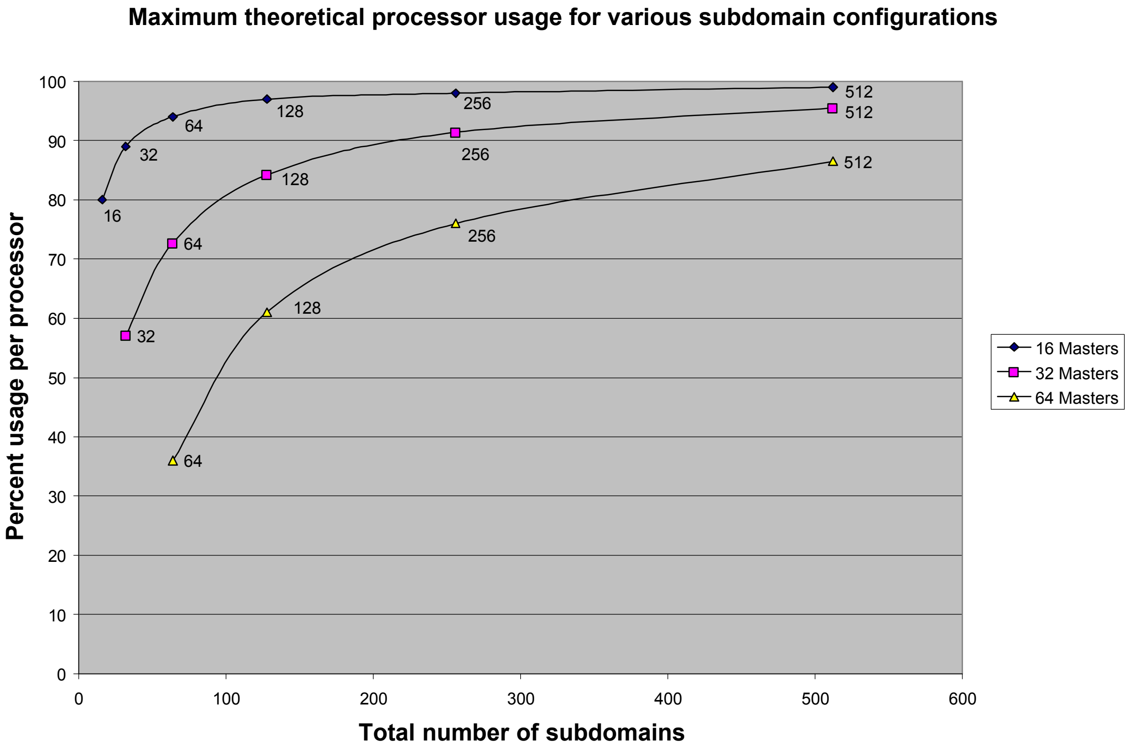


Figure 8. Graphical rendering of comparisons in Table 8

Actual timing figures for the above configurations agree fairly well with the predicted efficiencies, except that there is some degradation of performance with higher numbers of subdomains due to the inevitable overhead and communication costs. Nevertheless, we are able to exploit the massively parallel hardware rather well, and scaling studies have showed improved overall performance as problems are progressively doubled in size to eight times their original size. At 32 masters and 3000 processors we have not yet found in practice the point where scaling up a problem becomes counterproductive. Our principal limitation at the moment is the availability of computing nodes rather than our ability to exploit them effectively.

An example illustrating the effectiveness of subdomain overloading is given in Table 9 below. It shows that the theoretical efficiencies of Table 8 for 16 masters are in fact achieved in practice. This is an S_8 problem with 32 energy groups. It has fourfold symmetry about one axis, and thus is a good candidate for load balancing. We see from Table 9 that the product of the time and predicted theoretical efficiency is nearly constant as expected, and corresponds to a computation time of approximately 1200 seconds in the limiting case of 100% efficiency. In problems with no such symmetry, perfect load balancing is more difficult to achieve, and we can find that overall times remain nearly constant in spite of increasing theoretical efficiency. Moreover, if we attempt load balancing with too crude an overall mesh (thus resulting in few choices as to where to place partitions during the decomposition phase), then resorting to larger numbers of subdomains can be counterproductive. It follows that the opportunities for finer decomposition (and hence greater parallelism) occur when we model our problems with more and finer grids overall.

Table 9. Timing study for subdomain overloading efficiencies

Number of masters	Number of subdomains	Time (seconds)	Theoretical efficiency	Efficiency × time
16	16	1440*	.80	1150
16	32	1330*	.89	1184
16	64	1272*	.94	1196

* Due to the AMR nature of the code, slight variations in total zone count occurred between runs so these times have been normalized to a constant zone count.

We can now return to a problem to which we alluded earlier in section (4), namely the adjacency criterion as exemplified in Fig. 5. It turns out, occasionally, that our initial binary spatial decomposition may result in a configuration which fails that criterion. In such cases the question arises as to whether it would be better to abandon the scripting of sweeps which we developed in section (5) in favor of the greedy algorithm illustrated in Table 2, or alternatively to adjust our subdomain boundaries to fit the adjacency criterion at the expense of optimal load balancing. In fact, studies have been performed to find out how this trade-off could affect overall performance. The result is that satisfying the adjacency criterion is more important than optimizing the balance of work among the processors.

Acknowledgements

This work was performed under the auspices of the United States Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract Number W-7405-ENG-48.

References

IBM. See http://www.llnl.gov/asci/platforms/white (2000).

G. E. Sjoden and A. Haghighat, "Pentran: A Three-Dimensional Scalable Transport Code with Complete Phase-Space Decomposition," *Trans. Am. Nucl. Soc.*, Vol. 74, 181 (1996).

C. Aussourd, "An Adapted DSN Scheme for Solving the Two-Dimensional Neutron Transport Equation on a Structured AMR Grid," *Proc. Int. Conf. Mathematical Methods and Supercomputing for Nuclear Applications*, Saratoga Springs, New York, October 5-9, 1997, Vol. 1, p. 41, American Nuclear Society (1997).

M. J. Berger and P. Colella, "Local Adaptive Mesh Refinement for Shock Hydrodynamics," *Journal of Computational Physics*, Vol. 82, No. 1, pp. 64-84 (1989).

C. Aussourd, "Styx: A Multidimensional AMR Sn Scheme," Nuc. Sci. and Eng., Vol. 143, pp. 281-290 (2003).